US-PAT-NO: 5557798

DOCUMENT-IDENTIFIER: US 5557798 A

TITLE: Apparatus and method for providing decoupling of data exchange details for providing high performance communication between software processes

---------- KWIC ---------

Application Filing Date - AD (1):
19901221

Brief Summary Text - BSTX (17):
The data-exchange component of the communication interface according to the teachings of the invention includes a **forms-manager** module and a forms-class manager module. The **forms-manager** module handles the creation, storage, recall and destruction of instances of forms and calls to the various functions of the forms-class manager. The latter handles the creation, storage, recall, interpretation, and destruction of forms-class descriptors which are data records which record the format and semantic information that pertain to particular classes of forms. The forms-class manager can also receive requests from the application or another component of the communication interface to get a particular field of an instance of a form when identified by the name or meaning of the field, retrieve the appropriate form instance, and and deliver the requested data in the appropriate field. The forms-class manager can also locate the class definition of an unknown class of forms by looking in a known repository of such class definitions or by requesting the class definition from the forms-class manager linked to the foreign process which created the new class of form. Semantic data, such as field names, is decoupled from data representation and organization in the sense that semantic information contains no information regarding data representation or organization. The communication interface of the invention implements data decoupling in the semantic sense and in the data format sense. In the semantic sense, decoupling is implemented by virtue of the ability to carry out semantic-dependent operations. These operations allow any process coupled to the communications interface to exchange data with any other process which has data organized either the same or in a different manner by using the same field names for data which means the same thing in the preferred embodiment. In an alternative embodiment semantic-dependent operations implement an aliasing or synonym conversion facility whereby incoming data fields having different names but which mean a certain thing are either relabeled with field names understood by the requesting process or are used as if they had been so relabeled.

Detailed Description Text - DETX (43):
Format operations are performed by the **forms-manager** modules of the communications interface. For example, the first format operation in the

hypothetical transfer would be performed by the **forms-manager** module 86 in FIG. 1, while the second format operation in the hypothetical transfer would be performed by the **forms-manager** module in the data-exchange component 88.

Detailed Description Text - DETX (44):

Referring to FIG. 8, there is shown a flowchart of the operations performed by the **forms-manager** modules in performing format operations. Further details regarding the various functional capabilities of the routines in the **forms-manager** modules of the communications interface will be found in the functional specifications for the various library routines of the communications interface included herein. The process of FIG. 8 is implemented by the software programs in the **forms-manager** modules of the data-exchange components in the communications interface according to the teachings of the invention. The first step is to receive a format conversion call from either the application or from another module in the communications interface. This process is symbolized by block 90 and the pathways 92 and 94 in FIG. 1. The same type call can be made by the application 18 or the communications component 96 for the host computer 12 in FIG. 1 to the **forms-manager** module in the data-exchange component 88, since this is a standard functional capability or "tool" provided by the communication interface of the invention to all client applications. Every client application will be linked to a communication interface like interface 20 in FIG. 1.

Detailed Description Text - DETX (45):

Typically, format conversion calls from the communication components such as modules 30 and 96 in FIG. 1 to the **forms-manager** module will be from a service discipline module which is charged with the task of sending a form in format 1 to a foreign application which uses format 2. Another likely scenario for a format conversion call from another module in the communication interface is when a service discipline has received a form from another application or service which is in a foreign format and which needs to be converted to the format of the client application.

Detailed Description Text - DETX (46):

The format conversion call will have parameters associated with it which are given to the **forms manager**. These parameters specify both the "from" format and the "to" or "target" format.

Detailed Description Text - DETX (48):

The tables of FIGS. 9 and 10 probably would not be the only tables necessary for sending a form from the application 16 to the application 18 in FIG. 1. There may be further format-specific tables necessary for conversion from application 16 format to DEC machine format and for conversion from IBM machine format to application 18 format. However, the general concept of the format conversion process implemented by the **forms-manager** modules of the communications interface can be explained with reference to FIGS. 9, 10 and 11.

Detailed Description Text - DETX (49):

Assume that the first conversion necessary in the process of sending a form from application 16 to application 18 is a conversion from DEC machine format to a packed format suitable for transmission over an ETHERNET.TM. network. In this case, the format conversion call received in step 90 would invoke processing by a software routine in the **forms-manager** module which would perform the process symbolized by block 98.

Detailed Description Text - DETX (54):

Next, the process symbolized by block 104 in FIG. 8 is performed by accessing the general conversion procedures table shown in FIG. 11. This is a table which identifies the conversion program in the **forms manager** which performs the actual work of converting one primitive class of form to another primitive class of form. This table is organized with a single entry for every "from"--"to" format pair. Each entry in the table for a "from"--"to" pair includes the name of the conversion routine which does the actual work of the conversion. The process symbolized by block 104 comprises the steps of taking the "from"--"to" pair determined from access to the format-specific conversion table in step 102 and searching the entries of the general conversion procedures table until an entry having a "from"--"to" match is found. In this case, the third entry from the top in the table of FIG. 11 matches the "from"--"to" format pair found in the access to FIG. 9. This entry is read, and it is determined that the name of the routine to perform this conversion is ASCII.sub.-- ETHER. (In many embodiments, the memory address of the routine, opposed to the name, would be stored in the table.)

Detailed Description Text - DETX (61):

Returning to consideration of the preferred embodiment, such Get.sub.-- Field calls may be made by client applications directly to the forms-class manager modules such as the module 122 in FIG. 1, or they may be made to the communications components or **forms-manager** modules and transferred by these modules to the forms-class manager. The forms-class manager creates, destroys, manipulates, stores and reads form-class definitions.

Detailed Description Text - DETX (65):

After locating the field of interest in the class definition, the class manager returns a relative address pointer to the field of interest in instances of forms of this class. This process is symbolized by block 126 in FIG. 12. The relative address pointer returned by the class manager is best understood by reference to FIGS. 2, 4 and 6. Suppose that the application which made the Get.sub.-- Field call was interested in determining the age of a particular player. The Get.sub.-- Field request would identify the address for the instance of the form of class 1002 for player Blackett as illustrated in FIG. 6. Also included in the Get.sub.-- Field request would be the name of the field of interest, i.e., "age". The class manager would then access the instance of the form of interest and read the class number identifying the particular class descriptor or class definition which applied to this class of forms. The class manager would then access the class descriptor for class 1002 and find a class definition as shown in FIG. 4. The class manager would then access the class definitions for each of the fields of class definition 1002 and would compare the field name in the original Get.sub.-- Field request to

the field names in the various class definitions which make up the class definition for class 1002. In other words, the class manager would compare the names of the fields in the class definitions for classes 10, 1000, and 1001 to the field name of interest, "Age". A match would be found in the class definition for class 1000 as seen from FIG. 2. For the particular record format shown in FIG. 6, the "Age" field would be the block of data 62, which is the tenth block of data in from the start of the record. The class manager would then return a relative address pointer of 10 in block 126 of FIG. 12. This relative address pointer is returned to the client application which made the original Get.sub.-- Field call. The client application then issues a Get.sub.-- Data call to the **forms-manager** module and delivers to the **forms-manager** module the relative address of the desired field in the particular instance of the form of interest. The **forms-manager** module must also know the address of the instance of the form of interest which it will already have if the original Get.sub.-- Field call came through the **forms-manager** module and was transferred to the forms-class manager. If the **forms-manager** module does not have the address of the particular instance of the form of interest, then the **forms manager** will request it from the client application. After receiving the Get.sub.-- Data call and obtaining the relative address and the address of the instance of the form of interest, the **forms manager** will access this instance of the form and access the requested data and return it to the client application. This process of receiving the Get.sub.-- Data call and returning the appropriate data is symbolized by block 128 in FIG. 12.